4
MAC and HMAC

In this chapter, we will learn about **message authentication codes** (**MACs**), also known as **authentication tags**. MACs are used in popular secure network protocols, such as *TLS*, *SSH*, and *IPsec* in order to establish both the integrity and authenticity of the transmitted data. They are also used in proprietary network protocols, for example, in financial software, for the same purpose. Additionally, MACs can be used in non-networked authenticated encryption, as we demonstrated in *Chapter 2*, *Symmetric Encryption and Decryption*. Another application of MAC is as the basis of some key derivation functions, such as PBKDF2. Key derivation will be covered in more detail in *Chapter 5*, *Derivation of an Encryption Key from a Password*. We will learn how to calculate a MAC using both the command line and C code.

In this chapter, we are going to cover the following topics:

- What is a MAC?
- Understanding MAC function security
- HMAC – a hash-based MAC
- MAC, encryption, and the Cryptographic Doom Principle
- How to calculate HMAC on the command line
- How to calculate HMAC programmatically

Technical requirements

This chapter will contain commands that you can run on a command line along with C source code that you can also build and run. For the command-line commands, you will need the `openssl` command-line tool with OpenSSL dynamic libraries. To build the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement an example program in this chapter, in order to practice what we are learning. The full source code of that program can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter04`.

What is a MAC?

A MAC is a short array of bits, for example, 256 bits, that **authenticates** a message. **Message authentication** means that the receiver of the message can verify that the message is coming from the stated sender and has not been changed during the transfer. In order to generate a MAC, the sender needs a message and a **secret key**. In order to verify the MAC, the receiver requires the message and the same secret key. A MAC is produced by a **MAC function**.

The difference between a MAC and a message digest is that a message digest is not protected against forgery; however, a MAC does have such protection. If both the message and its digest are transmitted over an unprotected network, an attacker can change the message and recalculate its digest so that the changed digest will match the changed message. On the other hand, if a message is transmitted with its MAC, an attacker cannot recalculate the MAC for the changed message in the same way. This is because they do not possess the secret key. Therefore, it is widely agreed that message digests only provide message **integrity**, but MACs provide both **integrity** and **authenticity**.

In addition, MACs are different from **digital signatures**. Digital signatures use asymmetric cryptography so that the signer and the verifier of the signature use different keys of the same keypair. Only the signer can produce the signature because only they possess the **private key**. Hence, digital signatures provide **non-repudiation**, meaning that the signer cannot deny that they possessed information that they signed. When using MACs, both the sender and the receiver use the same secret key. Therefore, both the sender and the receiver can generate a MAC for any information that they possess, and it is difficult for a third party to determine which of them has generated the MAC. Hence, MACs, unlike digital signatures, do not provide non-repudiation. More information about digital signatures is given in *Chapter 7*, *Digital Signatures and Their Verification*.

Being different from both hash functions and digital signature functions, MAC functions also have different security requirements.

Understanding MAC function security

A MAC function is considered secure if it can resist the list of forgery attacks described as follows, which have been listed from more difficult to less difficult. All of the attacks that are described suggest that an attacker does not possess a secret key:

- A **universal forgery attack** is successful if an attacker can create a valid MAC for any message given to them.
- A **selective forgery attack** aims to produce the correct MAC for a particular message. The message is chosen prior to the attack and, usually, has some value for the attacker if they can authenticate it.
- An **existential forgery attack** works by finding a pair of any kind of message and its respective MAC.

- An **existential forgery under a chosen-message attack** suggests that there is an oracle that can generate a MAC for any message chosen by the attacker. The attacker can feed messages to the oracle, obtain MACs, and analyze the behavior of the oracle and the MAC function that the oracle performs. The attack is successful if the attacker can find a pair of any kind of message that was not fed to the oracle and its corresponding MAC.

Of course, as with other cryptographic attacks, any attack on a MAC can only be considered successful if it is executed without infeasible amounts of computation.

The security of a MAC function is measured using the same measurement units as the security of other cryptographic functions – in security bits. It is important to note that, usually, a MAC function uses another underlying function that has its own security properties. The security of the MAC function depends on the security of its underlying function and the secret key that was used.

There are different types of MAC functions, and the one that is used most often in secure network protocols is HMAC.

HMAC – a hash-based MAC

Hash-based Message Authentication Code (**HMAC**) is a MAC that is generated by the **HMAC function**.

The HMAC function uses a cryptographic hash function and a secret key. The function is not overly complex and can be understood by many non-cryptographers. It is defined as follows:

```
HMAC(K, message) = H(K' XOR opad || H(K' XOR ipad || message))
```

Here, the following can be understood:

- `Message`: This is the message to be authenticated.
- `H`: This is the hash function, for example, SHA3-256.
- `K`: This is the secret key.
- `K'`: This is a block-sized key derived from `K` depending on the hash function's internal block size, `B`.
- `ipad`: This is the inner padding, consisting of byte `0x36`, which is repeated `B` times.
- `opad`: This is the outer padding, consisting of byte `0x5C`, which is repeated `B` times.
- `||`: This refers to a concatenation.

Note that `K'` is derived from `K`, as follows:

- If the length of `K` is less than or equal to `B`, then `K'` is `K` padded with `0x00` bytes up to the length of `B`.
- If the length of `K` is greater than `B`, then `K'` is `H(K)` padded with `0x00` bytes up to the length of `B`.

Note that the hash function's internal block size, `B`, is not the same as the produced hash length. As a rule, the hash length is less than `B`. For example, the hash length of the SHA-256 function is 256 bits, but its internal block size is 1088 bits.

It is worth mentioning that an HMAC function using a particular hash function is often called by a name consisting of the *HMAC-* prefix and the underlying hash function name. For instance, an *HMAC* function using the *SHA-256* function is called *HMAC-SHA-256*.

An HMAC function produces an HMAC value of the same length as the hash length of the underlying hash function. For example, the HMAC-SHA-256 function produces a 256-bit HMAC.

A curious reader might ask the following: why do we need to complicate the HMAC function so much with nested hashing and padding? Why can't we just assume that the HMAC implementation is the following?

```
H(K || message)
```

Well, the short answer is that those complications are needed to counter various attacks on HMAC, which, first and foremost, includes the **length extension attack**. The length extension attack aims to produce a valid MAC by appending an attacker-controlled message to the original message, without knowing the original message and the secret key. Many cryptographic hash functions, including SHA-256, are vulnerable to the length extension attack. Newer hash functions families, such as SHA3, are not susceptible to the length extension attack and can be used with simpler MAC functions. For example, for SHA-3 functions, their own MAC function exists: **KECCAK Message Authentication Code** (**KMAC**).

When it comes to security strength, the security level of an uncompromised HMAC function can be calculated as follows:

```
SecurityLevel = min(K_bits, L)
```

Here, the following can be understood:

- `K_bits`: This is the security strength of secret key `K` in bits.
- `L`: This is the hash length of the underlying hash function in bits.

For example, if a 256-bit strong secret key is used, the HMAC-SHA-256 function achieves its highest security level: 256 bits.

So, what is the security strength of a secret key? Essentially, it is the key's entropy. If a key is generated by a cryptographically secure random generator, its security strength is the same as its length. This means that a 256-bit long key has 256-bit security strength. If the random generator that was used is suspected to have weak randomness, for example, every 2 generated bits only have 1 bit of entropy, then it makes sense to generate a key that is two times longer (512 bits) to get the same security strength (256 bits). But what if the key is derived from a low-entropy data source, such as a password? Well, the key cannot have more entropy than its source, and the security level of the whole HMAC function with such a key is degraded to the amount of entropy in the password. More information about how to derive keys from passwords is given in *Chapter 5*, *Derivation of an Encryption Key from a Password*.

An HMAC function can take a key of any length. However, keys shorter than `L`, considering that they are obtained from a cryptographically secure random generator, are strongly discouraged because they weaken the security of the HMAC function. Keys longer than `L` can be used, but a longer key does not improve the security of the HMAC function very much. This is because the output of the HMAC function is still limited to `L`. Therefore, the most obvious key length is just `L`. When using HMAC in the TLS protocol, OpenSSL uses `L`-long secret keys.

A secret key for the HMAC function must be infeasible to guess, just like a key for symmetric encryption. This means that the key must have enough entropy. One way to achieve that is to generate the key using a cryptographically strong pseudorandom generator.

As mentioned earlier, HMAC security depends on the security of the underlying hash function. However, an HMAC function is often significantly more secure than its underlying hash function if the latter becomes compromised. For example, even though the *MD5* hash function is severely compromised, and its current security level is 18 bits, the security level of the *HMAC-MD5* function is believed to be 97 bits. But even though an HMAC function is cryptographically stronger than its hash function, it makes sense to avoid any unnecessary risks and to use an HMAC function based on a secure hash function, for instance, HMAC-SHA-256.

One of the practical values of HMAC is that HMAC is the type of MAC that is used in the TLS, SSH, and IPsec protocols. In the next section, we will explain how a MAC can be combined with encryption and used in secure network protocols.

MAC, encryption, and the Cryptographic Doom Principle

When combining a MAC with encryption, one of the following schemes is used:

- **Encrypt-then-MAC** (**EtM**): Here, the plaintext is encrypted, then the MAC is calculated on the *ciphertext* and sent together with the ciphertext.
- **Encrypt-and-MAC** (**E&M**): Here, the plaintext is encrypted, but the MAC is calculated on the *plaintext* instead of the ciphertext. The ciphertext and the MAC are then sent together.
- **MAC-then-Encrypt** (**MtE**): Here, the MAC is calculated on the *plaintext*. Concatenation of the plaintext and the MAC is then encrypted.

On the receiving side, the EtM scheme allows you to check the authenticity of the encrypted message before the decryption operation. The other two schemes require decryption of the whole plaintext before the MAC can be verified.

Security researchers regard EtM as the most secure scheme, provided that a strong and unforgeable MAC function is used. From a security point of view, it makes sense to verify the data authenticity and discard bad messages as soon as possible, especially if the message is coming from an untrusted channel, such as the internet. If decryption must be done before the message authenticity is verified by the receiver, an attacker can change any messages in transmission and attack the encryption before the receiver can discard the message.

In 2011, a famous security researcher, Moxie Marlinspike, formulated the *Cryptographic Doom Principle*. Put simply, if you have to perform *any* cryptographic operation before verifying the MAC on a message you've received, it will *somehow* inevitably lead to doom. Moxie then mentions some examples when violating the Cryptographic Doom Principle makes it possible to mount successful attacks on encryption.

The principle sounds quite bold, especially the *any* and *inevitable* parts. In my opinion, it should be taken more as a general recommendation than a hard rule.

That being said, even though EtM is the most secure scheme, it is also possible to implement the other two schemes securely, albeit more care and attention are needed.

Until several years ago, authentication (MAC) and encryption could only be used as separate operations. However, recently, authenticated encryption modes, such as AES-GCM or ChaCha20-Poly1305, have become popular and are superseding non-authenticated encryption modes. Authenticated encryption modes provide authentication as one of their core features and do not require separate authentication operations, such as running an HMAC function on plaintext or ciphertext. If you are using authenticated encryption, you don't need to think about EtM versus MtE. There is no established term for it, but you could say that you are doing *MAC-while-Encrypt* when using authenticated encryption.

As you can see, there are different methods of combining authentication with encryption. But which methods are used in the popular security protocols?

Historically, *TLS* has been using the MtE scheme. The TLS protocol is based on the SSL protocol that was invented in 1995 when the superiority of the EtM scheme had not yet been proven. Later, the **EtM TLS protocol extension** was introduced, which allowed a TLS client and server to negotiate EtM instead of MtE. In TLS 1.2, authenticated encryption ciphers were added to the protocol. Finally, in TLS 1.3, only authenticated ciphers remained as the allowed ciphers. It's important to remember that even though authenticated ciphers do not require a separate MAC operation, HMAC is still used in TLS 1.3 as the basis for a **Pseudorandom Function** (**PRF**), which is needed for **key exchange**.

Similar to TLS, *SSH* is also an old protocol. The currently used version 2 of the SSH protocol was published in 1998. Historically, SSH has been using the E&Mscheme. However, the authentication scheme in SSH depends on the MAC algorithm, and newer MAC algorithms use the EtM scheme. The newer EtM algorithms are preferred over the older E&M algorithms. Also, the newer versions of the SSH implementations support authenticated ciphers that do not require a separate MAC.

IPsec has been using EtM since the beginning, as it was created in the 1990s. Later, authenticated ciphers were added to IPsec, similar to TLS and SSH.

We have now learned what MAC and HMAC are and how they are used. Let's do some practice and learn how to calculate HMAC on the command line.

How to calculate HMAC on the command line

Now we will learn how to calculate HMAC on the command line. To do this, we need to instruct the `openssl` tool on which type of MAC function to run and which underlying function to use. The MAC function will be `HMAC`, and for the underlying function, let's choose the `SHA-256` hash function.

The `openssl` tool provides two methods of HMAC calculation:

- Using the `openssl dgst` subcommand. This is the same subcommand that we used to calculate a message digest in the previous chapter.
- Using the `openssl mac` subcommand. A new subcommand was introduced in OpenSSL 3.0.

We will learn how to use both methods.

As usual, first, let's generate a sample file that we will use as input for HMAC calculation:

```
$ seq 20000 >somefile.txt
```

HMAC calculation requires a secret key. So, let's generate it:

```
$ openssl rand -hex 32
df036c471b612f8ad099078d8e3bd9c64339e7aeab56ec-
75e2222c415db113de
```

Here is how you calculate HMAC using the `openssl dgst` subcommand. In order to calculate HMAC instead of the message digest, you have to add `-mac HMAC` to the command-line switches and supply the secret key via the `-macopt hexkey:` switch:

```
$ openssl dgst -sha-256 -mac HMAC -macopt \
hexkey:df036c471b612f8ad099078d8e3bd9c64339e7aeab56ec-
75e2222c415db113de \
```

```
somefile.txt
HMAC-SHA256(somefile.txt)= 55e18ba91be755133ab0f4dbca5d06f2e7d-
f0b6bb4cd5f16f9f2d2f7cf83372c
```

As you can see, HMAC calculation on the command line is almost as easy as message digest calculation.

The following shows you how to do the same HMAC calculation using the new `openssl mac` subcommand:

```
$ openssl mac -digest SHA-256 –macopt \ hexkey:df036c471b-
612f8ad099078d8e3bd9c64339e7aeab56ec75e2222c415db113de \
-in somefile.txt \
HMAC
55E18BA91BE755133AB0F4DBCA5D06F2E7DF0B6BB4CD5F16F9F2D2F-
7CF83372C
```

Note that we have got the same HMAC value, but the output format is slightly different.

Now, let's learn how to calculate HMAC programmatically in a C program.

How to calculate HMAC programmatically

We are going to implement the `hmac` program that will calculate HMAC-SHA-256.

Our `hmac` program will need two command-line arguments:

1. The input filename
2. The secret key, hex-encoded

When it comes to the **Application Programming Interface** (**API**) for HMAC calculations, OpenSSL 3.0 provides three whole APIs:

- The deprecated legacy low-level API, consisting of functions with an `HMAC_` prefix. A fun fact about this API is that even though it's not an `EVP` API, it uses the `EVP_MD` API to access underlying message digest functions.
- The `EVP_DigestSign` API. This API is mostly used for digital signatures. It is possible to use this API for MAC, but such usage is not very intuitive.
- The `EVP_MAC` API. This new API was specially created for MAC and introduced in OpenSSL 3.0. We are going to use *this* API in our code.

The official documentation for the `EVP_MAC` API can be found on the man page of the same name:

```
$ man EVP_MAC
```

The following man pages could also be interesting:

```
$ man EVP_MAC-HMAC
$ man OSSL_PARAM
$ man OSSL_PARAM_int
```

As in the previous chapters, let's make a high-level implementation plan:

1. Fetch the HMAC algorithm implementation description as the `EVP_MAC` object.
2. Create MAC calculation parameters as an `OSSL_PARAM` array.
3. Create and initialize the MAC calculation context, `EVP_MAC_CTX`.
4. Read data from the input file, chunk by chunk, and feed the obtained data to the MAC calculation context.
5. Finalize the calculation and get HMAC.
6. Print HMAC to `stdout`.

Let's start coding!

Implementing the hmac program

Here is how we implement the `hmac` program:

1. First, we have to fetch the HMAC algorithm description from OpenSSL's default algorithm provider:

   ```
   EVP_MAC* mac = EVP_MAC_fetch(NULL, OSSL_MAC_NAME_HMAC,
   NULL);
   ```

   Pay attention to the `OSSL_MAC_NAME_HMAC` value passed to the function. That's how we specify that we want a particular HMAC algorithm, not another MAC algorithm, such as CMAC or GMAC.

2. Next, we have to create parameters for the MAC calculation context:

   ```
   OSSL_PARAM params[] = {
       OSSL_PARAM_construct_utf8_string(
           OSSL_MAC_PARAM_DIGEST,
           OSSL_DIGEST_NAME_SHA2_256,
           0),
       OSSL_PARAM_construct_end()
   };
   ```

Pay attention to the `OSSL_MAC_PARAM_DIGEST` and `OSSL_DIGEST_NAME_SHA2_256` values. That's how we specify that we want to use SHA-256 as the underlying message digest function.

3. We have fetched the algorithm and created the parameters; let's use them to create and initialize the MAC calculation context:

```
EVP_MAC_CTX* ctx = EVP_MAC_CTX_new(mac);
EVP_MAC_init(ctx, key, key_length, params);
```

We have also set the secret key and its length into the context. We have got that secret key as a command-line argument and unhexified it.

4. The initialization is complete. Now we need to feed data from the input file to the context:

```
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_
file);
    EVP_MAC_update(ctx, in_buf, in_nbytes);
}
```

5. After all the data has been fed to the context, finalize the calculation and get the resulting HMAC:

```
const size_t HMAC_LENGTH = 256 / 8;
unsigned char hmac[HMAC_LENGTH];
size_t out_nbytes = 0;
EVP_MAC_final(ctx, hmac, &out_nbytes, HMAC_LENGTH);
```

6. Now we no longer need the `EVP_MAC` and `EVP_MAC_CTX` objects. Let's free them and avoid any memory leaks:

```
EVP_MAC_CTX_free(ctx);
EVP_MAC_free(mac);
```

7. Finally, let's print the calculated HMAC in the same format as the `openssl mac` subcommand:

```
for (size_t i = 0; i < HMAC_LENGTH; ++i) {
    printf("%02X", hmac[i]);
}
printf("\n");
```

The complete source code of our `hmac` program can be found on GitHub as a `hmac.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter04/hmac.c`.

Running the hmac program

Let's run our HMAC calculation program and check whether it works:

```
$ ./hmac somefile.txt df036c471b612f8ad099078d8e3bd9c
64339e7aeab56ec75e2222c415db113de
55E18BA91BE755133AB0F4DBCA5D06F2E7DF0B6BB4CD5F16F9F2D2F-
7CF83372C
```

As you can see, the HMAC calculated by our small program is the same as the one calculated by the `openssl` tool in the previous section. Such an observation suggests that our program works correctly.

Summary

In this chapter, we learned what MACs are and how they differ from message digests and digital signatures. We also learned about MAC function security and attacks that a good MAC function must resist. Following this, we learned what an HMAC function is and what its security depends on. We finished the theoretical part with a review of several methods of combining a MAC function with encryption, discovered what the best method is, and discussed the Cryptographic Doom Principle.

In the practical part, we learned about two methods of HMAC calculation on the command line. Then, we also learned how to calculate HMAC programmatically in C code. We compared the resulting HMACs calculated by all the methods used and, to our satisfaction, confirmed that all the methods produced the same HMAC.

In the next chapter, we will learn about **Key Derivation Functions** (**KDFs**) and how to derive encryption keys from passwords.